NASA-TM-108129

7N-61-TM
136175
P.9

GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT: BACKGROUND AND EXPERIENCE

21st Hawaii International Conference on System Sciences
January, 1988

Ed Seidewitz
Code 554 / Flight Dynamics Analysis Branch

Goddard Space Flight Center
Greenbelt MD 20771
(301) 286-7631

Abstract

The effective use of Ada™ requires the adoption of modern software-engineering techniques such as object-oriented methodologies. A Goddard Space Flight Center Software Engineering Laboratory Ada pilot project has provided an opportunity for studying object-oriented design in Ada. The project involves the development of a simulation system in Ada in parallel with a similar FORTRAN development. As part of the project, the Ada development team trained and evaluated object-oriented and process-oriented design methodologies for Ada. Finding these methodologies limited in various ways, the team created a general object-oriented development methodology which they applied to the project. This paper discusses some background on the development of the methodology, describes the main principles of the approach and presents some experiences with using the methodology, including a general comparison of the Ada and FORTRAN simulator designs.

1. Introduction

Increased productivity and reliability from using Ada must come from innovative application of the non-traditional features of the language. However, past experience has shown that traditional development methodologies result in Ada systems that "look like a FORTRAN design" (see, for example, [Basili 85]). *Object-oriented* techniques provide an alternative approach to effective use of Ada. As the name indicates, the primary modules of an object-oriented design are *objects* rather than traditional functional procedures. Whereas a procedure models an action, an object models some entity in the problem domain, encapsulating both data about that entity and operations on that data. Ada is especially suited to this type of design because its package facility directly supports the construction of objects.

Ada is a registered trademark of the US Government (AJPO)
PAMELA is a registered trademark of George W. Cherry.

The Goddard Space Flight Center Software Engineering Laboratory is currently involved in an Ada pilot project to develop a system of about 50,000 statements [Nelson 86]. This project has provided an opportunity to explore object-oriented software development methods for Ada. The pilot system, known as "GRODY", is an attitude dynamics simulator for the Gamma Ray Observatory (GRO) spacecraft and is based on the same requirements as a FORTRAN system being developed in parallel.

The GRODY team was initially trained both in the Ada language and in Ada-oriented design methodologies. The team specifically studied the methodology promoted by Grady Booch [Booch 83] and the PAMELA™ methodology of George Cherry [Cherry 85]. Following this, during a training exercise, the team also began synthesizing a more general approach to object-oriented design. At an early stage of the GRODY development effort, the team produced high-level designs for GRODY using each of these methodologies.

Section 2 summarizes the comparison of methodologies made by the GRODY team. Section 3 then discusses in more detail the methodology which was actually used to develop the full GRODY design. Section 4 describes the resulting Ada design and compares it to the traditional FORTRAN design. Finally, section 5 provides some concluding lessons-learned and recommendations.

2. Comparison of Methodologies

This section presents the comparison made by the GRODY team of the Booch methodology, PAMELA and the general methodology developed by the team itself. All these methodologies address two basic questions: "How is the system design represented?" and "How is the design derived from requirements?"

2.1 Booch's Methodology

Grady Booch is, perhaps, the most influential advocate of object-oriented design in the Ada community [Booch 86b, Booch 87]. As learned by the GRODY team, Booch's methodology derives a design from a textual specification or informal design [Booch 83]. The technique is to underline all the nouns and verbs in the specification. The objects in the

(NASA-TM-108129) GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT: BACKGROUND AND EXPERIENCE (NASA) 9 p

N93-70957

Unclas

Z9/61 0136175

sign derive from the nouns; object operations derive from verbs. Obviously, some judgment must be used to regard irrelevant nouns and verbs and to translate the naining concepts into design objects. Once the objects have en identified, the design can then be represented igrammatically using a notation which shows the pendencies between Ada packages and tasks which plement the objects.

ie Booch design methodology contains all the basic amework of the object-oriented approach. However, plication of this methodology to GRODY indicated that it s not readily applicable to sizable systems. The team found graphical notation clear but not detailed or rigorous ough. Further, Booch gives no explicit method for gramming a hierarchical decomposition of objects, which is eded for any sizable system. Booch's notation does not, erefore, seem to be a complete design notation. Note, wever, that in more recent work Booch has extended the pe of the notation to address some of these shortcomings ooch 87].

second difficulty of Booch's methodology is in the technique deriving the design from the specification text. This works ll when the specification can be written concisely in a few ragraphs. However, when the system requirements are large, with GRODY, this can be difficult. In addition, any empt to use such a technique directly on a requirements cument such as ours is doomed to failure due to the sheer e and complexity of the document. Realizing such twbacks, Booch no longer advocates the use of this textual thod, which was never actually intended for large systems velopment [Booch 86b]. Instead, he derives an object-ented design from a data flow diagram based specification ooch 86a, Booch 87]. However, from the published amples it is still unclear how to systematically apply this thod to realistic systems.

PAMELA

e second methodology considered by the GRODY team was Process Abstraction Method for Embedded Large plications (PAMELA) developed by George Cherry herry 85, Cherry 86]. PAMELA is oriented toward real-ne and embedded systems. PAMELA is *process-oriented*, so PAMELA design consists of a set of interacting *concurrent* cesses. A well designed *process* is effectively a concurrent ject, thus PAMELA is object-oriented in a general way.

MELA uses a powerful graphical notation without the twbacks found in Booch's notation [Cherry 86]. During the MELA design processes, the designer successively mposes processes into concurrent subprocesses until he ches the level of primitive *single-thread* processes. The ODY team found that PAMELA provides fairly explicit uristics for constructing good processes. The designer uses se hints to construct the top-level processes from the system cification. The designer then recursively decomposes each n-primitive processes until only primitive processes remain. e primitive processes can then be coded as Ada tasks with a single thread of control. Non-primitive processes are simply packages of lower level processes and thus contain multiple threads of control.

PAMELA's heuristics can be very effective when designing a real-time system that is heavily driven by external asynchronous actions. In other cases, however, they require considerable interpretation to be applicable. Although parts of GRODY might conceptually be concurrent (because GRODY simulates actions that happen in parallel in the real world), there is no requirement for concurrency in the simulation of these actions because GRODY does not have to interface with any active external entity (except the user). In addition, since GRODY runs on a sequential machine, the overhead of Ada tasking and rendezvous could greatly degrade the time performance of the system. Thus, one interpretation of PAMELA's principles might leave very large sections of GRODY as primitive single-thread processes, with only a few concurrent objects in the entire design. To proceed further in the decomposition, the designer has to rely more on intuition about what makes a good object and rely less on the methodology. In fact, at the time that the GRODY team was using PAMELA, it provided no support for the decomposition and design of anything below the level of the primitive process, an Ada task [Cherry 85]. Since then, Cherry has added several concepts to the methodology, including the use of abstract data types [Cherry 86]. However, the methodology remains weak for systems with a small amount of concurrency which are still to be designed in an object-oriented fashion.

2.3 General Object-Oriented Development

As a result of the above experiences, the GRODY team developed its own object-oriented methodology which attempts to capture the best points of the object-oriented approaches studied by the team as well as traditional structured methodologies [Seidewitz 86a, Seidewitz 87b, Stark 87]. It is designed to be quite general, giving the designer the flexibility to explore design alternatives easily. It is also based on principles that guide the designer in constructing good object-oriented designs. This methodology was used to develop the complete detailed design for GRODY.

This general object-oriented development ("GOOD") methodology is based on general principles of abstraction, information hiding and design hierarchy discussed in the next section. These principles are less explicit than Booch's methodology or PAMELA, but they do provide a firm paradigm for generating and evaluating an object-oriented design. Indeed, as mentioned above, the team found the Booch and PAMELA design construction techniques restrictive, often necessitating the designer to rely on intuition for object-oriented design. The GOOD methodology is an attempt to codify this intuition into a basic set of principles that provide guidance while leaving the designer the flexibility to explore various design approaches.

In addition, we have also considered, independently of Booch, the transition from structured analysis [DeMarco 79] to object-oriented design in the context of the GOOD methodology, developing a technique known as *abstraction analysis* [Seidewitz 86a, Seidewitz 86b]. This technique is analogous to transform and transaction analysis used in structured design [Yourdon 78]. However, proceeding into object-oriented design from a structured analysis, by whatever means, requires an "extraction" of problem domain entities from traditional data flow diagrams. From an object-oriented viewpoint, it seems appropriate to instead *begin* a specification effort by identifying the entities in a problem domain and their interrelationships. Study is continuing on including such object-oriented system specification techniques in the GOOD methodology and on applying object-oriented principles throughout the Ada life cycle [Stark 87].

3. The GOOD Methodology

As a result of the comparison discussed in section 2, the GRODY team decided to apply the GOOD methodology to the full GRODY design. This section provides an overview of the principles and notation used during the GRODY design.

3.1 Designing with Objects

The intent of an object is to represent a problem domain entity. The concept of *abstraction* deals with how an object presents this representation to other objects [Booch 86b, Dijkstra 68]. Ideally, the objects in a design should directly reflect the problem domain entities identified during system specification. However, various design considerations may require splitting or grouping of objects and there will almost always be additional objects in a design to handle "executive" and "utility" functions. Thus there is a spectrum of levels of abstraction of objects in a design, from objects which closely model problem domain entities to objects which really have no reason for existence [Seidewitz 86b]. The following are some points in this scale, from strongest to weakest:

Entity Abstraction - An object represents a useful model of a problem domain entity or class of entities.

Action Abstraction - An object provides a generalized set of operations which all perform similar or related functions (this is similar to the idea of a "utility" object in [Booch 87]).

Subsystem Abstraction - An object groups together a set of objects and operations which are all related to a specific part of a larger system (this is similar to the "subsystem" concept in [Booch 87]).

The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of *information hiding* states that such details should be kept secret from other objects [Booch 87, Parnas 79], so as to better preserve the abstraction modeled by the object.

The principles of abstraction and information hiding provide the main guides for creating "good" objects. These objects must then be connected together to form an object-oriented design. This design is represented using a graphical *object diagram* notation [Seidewitz 86b]. Similarly to Booch's notation, object diagrams show control flow and module dependencies between objects. However, they can be hierarchically leveled as with PAMELA's process graphs.

3.2 Design Hierarchies

The construction of an object-diagram-based design is mediated by consideration of two orthogonal hierarchies in software system designs [Rajlich 85]. The *composition* hierarchy deals with the composition of larger objects from smaller component objects. The *seniority* hierarchy deals with the organization of a set of objects into "layers". Each layer defines a *virtual machine* which provides services to senior layers [Dijkstra 68]. A major strength of object diagrams is that they can distinctly represent these hierarchies.

The composition hierarchy is directly expressed by *leveling* object diagrams (see figure 1). At its top level, any complete system may be represented by a single object which interacts with *external objects*. Beginning at this system level, each object can then be refined into component objects on a lower-level object diagram, designed to meet the specification for the object. The result is a leveled set of object diagrams which completely describe the structure of a system. At the lowest level, objects are completely decomposed into *primitive objects* such as procedures and internal state data stores. At higher levels, object diagram leveling can be used in a manner similar to Booch's "subsystems" [Booch 87].

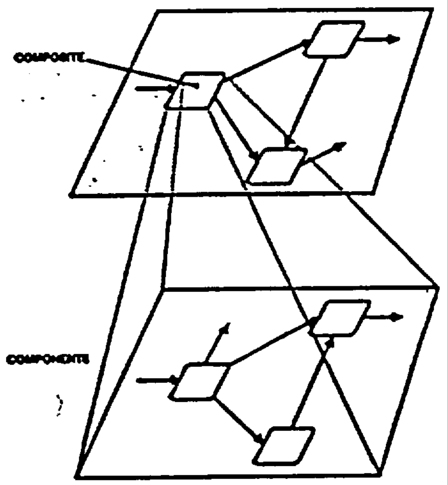

FIGURE 1 Composition Hierarchy

The seniority hierarchy is expressed by the topology of connections on a single object diagram (see figure 2). An arrow between objects indicates that one object calls *one or more* of the operations provided by another object. Any layer in a seniority hierarchy can call on any operation in junior layers, but *never* any operation in a senior layer. Thus, all relationships between objects must be contained within al machine layer. Object diagrams are drawn with the ty hierarchy shown vertically. Each senior object can be ed as if the operations provided by junior layers were ive operations" in an extended language. Each virtual e layer will generally contain several objects, each ted according to the principles of abstraction and ation hiding.

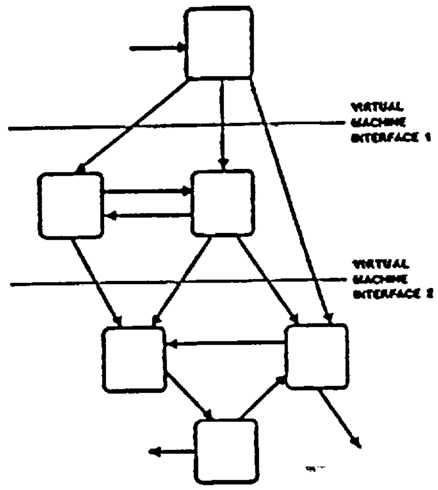

FIGURE 2 Seniority Hierarchy

signing Systems

ain advantage of a seniority hierarchy is that it reduces upling between objects. This is because all objects in one l machine layer need to know nothing about senior . Further, the centralization of the procedural and data control in senior objects can make a system easier to stand and modify.

ver, this very centralization can cause a messy bottleneck. h cases, the complexity of senior levels can be traded off st the coupling of junior levels. The important point is he strength of the seniority hierarchy in a design can be n from a *spectrum* of possibilities, with the best design ally lying between the extremes. This gives the designer power and flexibility in adapting system designs to fic applications.

example, consider a simplified attitude dynamics ation system similar to GRODY. The *attitude* of a craft is its orientation relative to inertial space, and an de dynamics simulator models the *rotational* motion of pacecraft in response to external disturbances and the craft control system. The problem domain for such a n includes the external environment, thrusters to control pacecraft, sensors to determine the current attitude, etc.

These entities interact with the spacecraft control system in a control loop outlined in figure 3.

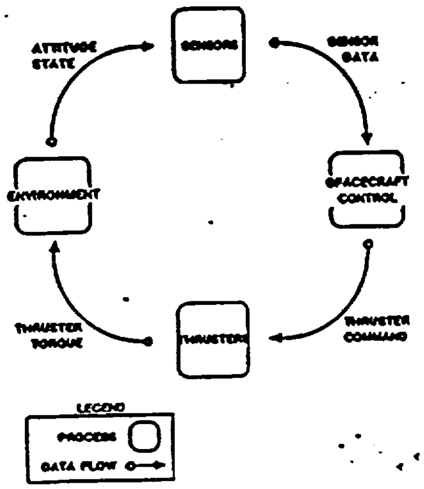

FIGURE 3 Attitude Dynamics Problem Domain

Figure 4 shows one possible preliminary design for the ATTITUDE SIMULATOR. For simplicity, the sensors and thrusters are represented by a single "SPACECRAFT HARDWARE" object in figure 4. Note that, by convention, the arrow labeled "RUN" is the initial invocation of the entire system. In preliminary design diagrams such as figure 4, it is sometimes convenient to show what data flows along certain control arrows, much in the manner of structure charts [Yourdon 78] or "Buhr charts" [Buhr 84]. These annotations will not appear on the final object diagrams.

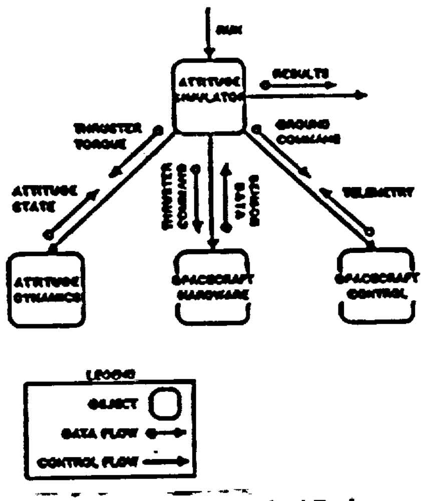

FIGURE 4 Centralized Design

ORIGINAL PAGE IS
OF POOR QUALITY

In figure 4, the junior level components do not interact directly. All data flow between junior level objects must pass through the senior object, though each object still receives and produces all necessary data (for simplicity not all data flow is shown in figure 4). This design is somewhat like an object-oriented version of the structured designs of Yourdon and Constantine [Yourdon 78].

We can remove the data flow control from the senior object and let the junior objects pass data directly between themselves, using operations within the virtual machine layer (see figure 5). The senior object has been reduced to simply activating various operations in the virtual machine layer, with very little data flow.

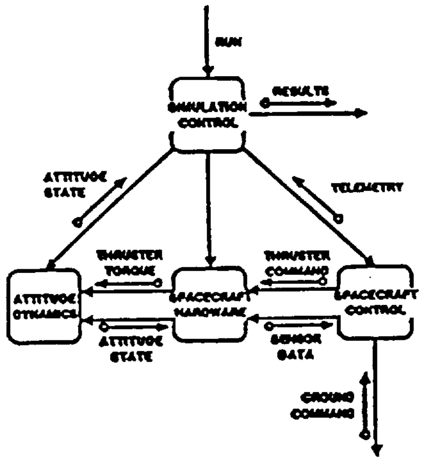

FIGURE 5 Design with Decentralized Data Flow

We can even remove the senior object completely by distributing control among the junior level objects (see figure 6). The splitting of the RUN control arrow in figure 6 means that the three objects are activated *simultaneously* and that they run *concurrently*. The seniority hierarchy has collapsed, leaving a *homologous* or non-hierarchical design [Yourdon 78] (no *seniority* hierarchy, that is; the composition hierarchy still remains)

A design which is decentralized like figure 6 at all composition levels is very similar to what would be produced by the PAMELA methodology [Cherry 86]. In fact, it should be possible to apply PAMELA design criteria to the upper levels of an object diagram based design of a highly concurrent system. All concurrent objects would then be composed, at a certain level, of objects representing certain process "idioms" [Cherry 86]. Below this level concurrency would generally no longer be advantageous.

To complete the design, we need to add a virtual machine layer of utility objects which preserve the level of abstraction of the problem domain entities. In the case of the ATTITUDE SIMULATOR these, objects might include VECTOR, MATRIX, GROUND COMMAND and simulation PARAMETER types. Figure 7 shows how these objects might be added to the simulator design of Figure 4.

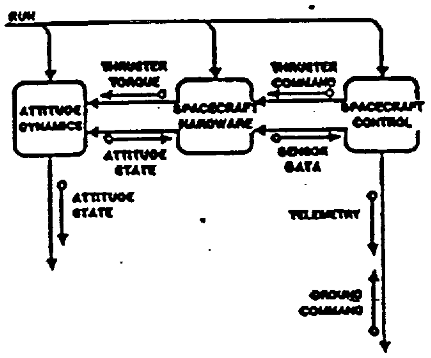

FIGURE 6 Decentralized Design

Figure 7 gives one complete level of the design of the ATTITUDE SIMULATOR. Note that figure 7 does not include the data flow arrows used in earlier figures. When there are several control paths on a complicated object diagram, it rapidly becomes cumbersome to show data flows. Instead, *object descriptions* for each object on a diagram provide details of the data flow.

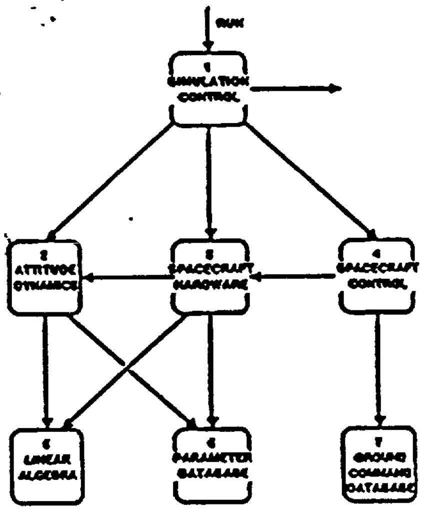

FIGURE 7 Attitude Dynamics Simulator Design

An object description includes a list of all operations provided by an object and, for each arrow leaving the object, a list of operations used from another object. We can identify the operations provided and used by each object in terms of the specified data flow and the designed control flow. The object description can be produced by matching data flows to operations. For example, the description for the ATTITUDE DYNAMICS object in figure 7 might be:

```
Provides:
 procedure Initialize;
 procedure Integrate (For_Duration: in DURATION);
 procedure Apply (Torque: in VECTOR);
 function Current_Attitude return ATTITUDE;
 function Current_Angular_Velocity
 return VECTOR;

Uses:
 5.0 LINEAR ALGEBRA
  Add (Vector)
  Dot
  Multiply (Scalar)
  Multiply (Matrix)

 6.0 PARAMETER DATABASE
  Get
```

We could next proceed to refine the objects used in figure 7 and recursively construct lower level object diagrams. These lower level designs must meet the functionality of the system specification and provide the operations listed in the object description. The design process continues recursively until the entire system is designed and all objects are completely decomposed.

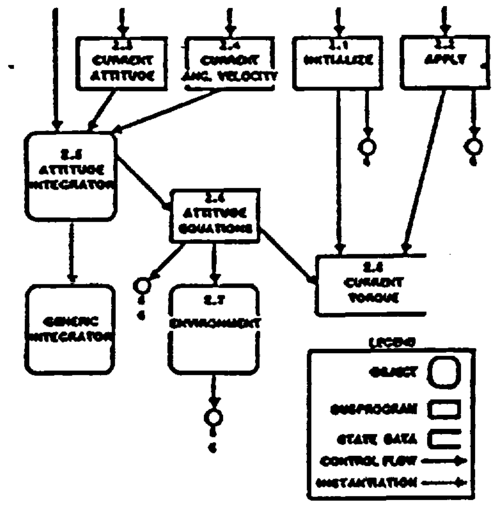

FIGURE 8 Attitude Dynamics Object Composition

For example, figure 8 shows the composition of the ATTITUDE DYNAMICS object. The component object ATTITUDE INTEGRATOR is the instantiation of a generic INTEGRATOR object which takes the function to be integrated as a generic parameter. The generic object is instantiated in figure 8 with the ATTITUDE EQUATION subprogram as the generic actual parameter. Most of the ATTITUDE DYNAMICS operations are shown in figure 8 as component procedures, represented by rectangles. The "Integrate" operation, however, is directly *inherited* from the ATTITUDE INTEGRATOR object.

3.4 Implementation

The transition from an object diagram to Ada is straightforward. Package specifications are derived from the list of operations provided by an object. For the ATTITUDE DYNAMICS object the package specification is:

```
package Attitude_Dynamics is

  subtype ATTITUDE is Linear_Algebra.MATRIX;

  procedure Initialize;
  procedure Integrate
    ( For_Duration : in DURATION );
  procedure Apply
    ( Torque : in Linear_Algebra.VECTOR );

  function Current_Attitude
    return ATTITUDE;
  function Current_Angular_Velocity
    return Linear_Algebra.VECTOR;

end Dynamics;
```

The package specifications derived from the top level object diagram can either be made library units or placed in the declarative part of the top level Ada procedure. For lower level object diagrams the mapping is similar, with component package specifications being nested in the package body of the composite object. States are mapped into package body variables. This direct mapping produces a highly nested program structure. Alternatively, some or all of these packages can be made library units or even reused from an existing library. However, this may require additional packages to contain data types and state variables used by two or more library units.

The process of transforming object diagrams to Ada is followed down all the object diagram levels until we reach the level of implementing individual subprograms. Low-level subprograms can be designed and implemented using traditional functional techniques. They should generally be coded as subunits, rather than being embedded in package bodies.

As mentioned in subsection 3.3, Attitude_Dynamics inherits its "Integrate" operation from a component object. Smalltalk's subclassing [Goldberg 83] provides an elegant means of supporting inheritance. Ada does not directly support inheritance, but the concept can be simulated by using "call-throughs." A call-through is a subprogram that has little function other than to call on another package's subprogram. To simulate inheritance when implementing the Attitude_Dynamics package the subprogram Integrate would be respecified in the Attitude_Dynamics package, with the subprogram body in Attitude_Dynamics calling on the corresponding operation from Attitude_Integrator.

This technique is clearly less elegant than Smalltalk subclassing, but it also has advantages. First, Ada allows inheritance from more than one object. Second, Smalltalk forces the inheritance of *all* operations and data. An operation can be overridden, but not removed, from a class. The Ada specification of the composite package gives the developer precise control over which operations and data items are visible or accessible. (See [Seidewitz 87] for a more detailed discussion of Ada and the concept of inheritance.)

The clear definition of abstract interfaces in an object-oriented design can also greatly simplify testing. When testing an object, there is a well defined "virtual machine" of operations it requires from objects at a junior level of abstraction, some of which may be stubbed-out for initial testing. Further, object-oriented composition encourages incremental integration testing, since the "unit testing" of a composite object really consists of "integration testing" the component objects at a lower level of abstraction.

4. Application to GRODY

As part of the GRODY project, a detailed assessment has been made of the team's experiences during design [Godfrey 87]. At this time, however, most of the observations must remain qualitative. Nevertheless, it is clear that the GRODY design is significantly different from previous FORTRAN simulator designs [Agresti 86].

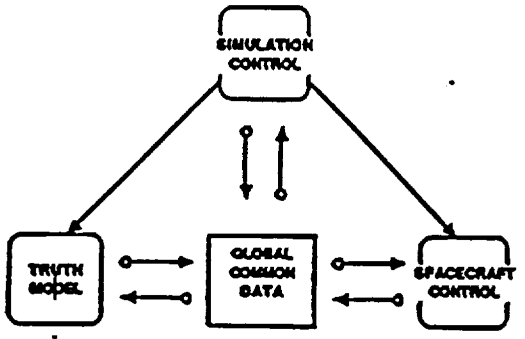

FIGURE 9 FORTRAN Simulator Design

4.1 Design Comparison

The design of the FORTRAN simulator has a strong heritage in previous simulator and ground support system designs. It consists of three major subsystems which interact as shown in figure 9. The "TRUTH MODEL" subsystem includes models of the spacecraft hardware, the external environment and the attitude dynamics; that is, the "real world" as opposed to the spacecraft control system. The SIMULATION CONTROL subsystem alternatively activates the SPACECRAFT CONTROL and TRUTH MODEL subsystems in a cyclic fashion. Data flow between subsystems, as well as system parameterization, is entirely though a set of global COMMON areas.

Since GRODY was derived from the same basic requirements as the FORTRAN design, there are similarities in the designs of the two systems. However, there are also some fundamental differences in the GRODY design that can be traced to the object- oriented methodology. Figure 10 is an object diagram of the main part of the GRODY design. This design is similar to the example design of figure 7. However, the GRODY team chose to combine the ATTITUDE DYNAMICS and SPACECRAFT HARDWARE objects into a single TRUTH MODEL object, similar to the corresponding subsystem in the FORTRAN design. Further, in GRODY the LINEAR ALGEBRA functions are part of a UTILITIES module not shown in figure 10.

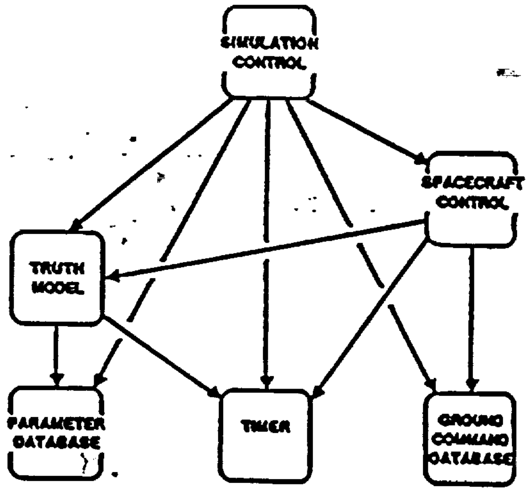

FIGURE 10 Ada Simulator Design

Unlike the FORTRAN design, consideration of the seniority hierarchy in the GRODY design led the GRODY team to place the TRUTH MODEL at a level junior to the SPACECRAFT CONTROL. The TRUTH MODEL is thus effectively passive, with the SPACECRAFT CONTROL calling on operations as needed to obtain sensor data and activate actuators. All sensor and command data is passed using these operations.

The simulation timing of GRODY is also different from the FORTRAN design. The object-oriented methodology led to consideration of a "TIMER" object in GRODY which provides an abstraction of the simulation time. This utility object provides a common time reference for the SPACECRAFT CONTROL and TRUTH MODEL separate from the SIMULATION CONTROL loop. Unlike the FORTRAN design, in GRODY the "cycle times" of the SPACECRAFT CONTROL and TRUTH MODEL are not the same. The GRODY team chose to faithfully model, in the SPACECRAFT CONTROL abstraction, the timing of the actual spacecraft control software, which is not under user control. However, GRODY allows the simulation user to set the cycle time for the TRUTH MODEL over a fairly wide range, to allow the user to trade-off speed and accuracy as desired.

Finally, the PARAMETER DATABASE and GROUND COMMAND DATABASE objects encapsulate user settable parameters for the simulation. Similar data is contained in COMMON blocks in the FORTRAN design. This encapsulation of "global" data is typical of object-oriented designs. It provides both increased protection of the data encapsulated and increased opportunity for reuse. For example, the simulation parameters in the FORTRAN design are COMMON block parameters which must be hard-coded into the user interface code. (For simplicity the user interface modules have not been included in the design diagrams here.) In the GRODY design, simulation parameters are identified by enumeration constants, which allows the user interface displays to be parameterized by external data files. This should greatly increase the reusability of the user interface.

4.2 Experience with the Methodology

The differences discussed above could probably have been incorporated into the FORTRAN design. However, it was largely the influence of the object-oriented approach which lead to their consideration for GRODY when they had not been considered in several previous designs of simulators for FORTRAN. Considerations of encapsulation and reusability indicate that the GRODY design may be "better" than the FORTRAN design. This is, of course, the goal of object-oriented methods. However, the true test of the merits of the GRODY design will only come from continuing studies of the comparative maintainability of the FORTRAN and Ada simulators.

n terms of the methodology itself, the team found the object diagram notation extremely useful for discussing the design during development. Further, the notation provided complete documentation of the design and was tailored specifically towards Ada. This made the transition to coding very smooth, and allowed the documentation to be readily updated as coding proceeded. By the end of coding, there were no major changes n the design and most changes that did occur were additions rather than alterations.

he object diagram notation evolved considerably during the RODY project in response to continuing experience with its use. The lack of a specific methodology at the start of the GRODY project was a problem for the team, as was the continuing evolution of the methodology over the duration of the project. Further, the fact that managers were not familiar with the new methodology made the use of object diagrams difficult at reviews. Another problem was that the detail of the object diagrams and the emphasis on keeping the documentation up-to-date required a lot of effort for maintaining a rather large design notebook. The team clearly saw the great need for automated tools to support the methodology in this area. Consideration is also being given on how to extend the object diagram notation to better cover such topics as generics, abstract data types and large system components.

5. Conclusion

The GRODY project has provided an extremely valuable experience in the application of object-oriented principles to a real system. This experience guided the creation of the GOOD methodology which is now being used on an increasing number of projects inside and outside of the Goddard Space Flight Center. As with any pilot project, some of the major products of GRODY are the lessons learned along the way. Some specific points on the methodology used in GRODY are [Godfrey 87]:

- The design methodology should be chosen as early as possible so that the team can be trained in this methodology and so that time will not be wasted trying to use an unsuitable methodology.

- The methodology chosen must exploit important Ada features such as packages, tasks and generics.

- Object diagrams were a very suitable representation for the GRODY design.

- The GOOD methodology seems to be an extremely useful method for system design.

- Compilable design elements developed in Ada are very useful for providing validation of the design as well as for documentation.

It also became clear during the GRODY project that the GOOD methodology does not fit comfortably into the traditional life cycle management model. At the very least, the design phase should be extended and design reviews should occur at different points in the life cycle. The preliminary design review should occur later in the design phase and should include detailed object diagrams for the upper levels of the system, perhaps down to the level at which the design becomes more procedural than object-oriented. The critical design review would then include the detailed procedural designs, perhaps using an Ada-based design language. This review might actually take place as a series of incremental reviews of different portions of the design. This later approach is supported by the well-defined modularity of an object-oriented design.

The traditional functional viewpoint provides a comprehensive framework for the entire software life-cycle. This viewpoint reflects the action-oriented nature of the machines on which software is run. The object-oriented viewpoint, however, reflects the natural structure of the problem domain rather than the implicit structure of our hardware. Thus, it provides a "higher-level" approach to software development which decreases the distance between problem domain and software solution. By making complex software easier to understand, this simplifies both system development and maintenance. Our experience with GRODY forms the basis for fruitfully applying this approach to future Ada projects.

References

[Agresti 86]
Agresti, William W., et. al. "Designing with Ada for Satellite Simulation: a Case Study," Proceedings of the 1st International Conference on Ada Applications for the Space Station, June 1986.

[Basili 85]
Basili, V. R., et. al. "Characterization of an Ada Software Development," Computer, September 1985.

[Booch 83]
Booch, Grady. Software Engineering with Ada, Benjamin/Cummings, 1983.

[Booch 86a]
Booch, Grady. "Object-Oriented Software Development," IEEE Transactions on Software Engineering, February 1986.

[Booch 86b]
Booch, Grady. Software Engineering with Ada, 2nd Edition, Benjamin/Cummings, 1986.

[Booch 87]
Booch, Grady. Software Components with Ada, Benjamin/Cummings, 1987.

[Buhr 84]
Buhr, R. J. A. System Design with Ada, Prentice-Hall, 1984.

[Cherry 85]
Cherry, George W. PAMELA Course Notes, Thought**Tools, 1985.

[Cherry 86]
Cherry, George W. PAMELA Designer's Handbook, Thought**Tools, 1986.

[Dijkstra 68]
Dijkstra, Edsgar W. "The Structure of the 'THE' Multiprogramming System," Communications of the ACM, May 1968.

[Godfrey 87]
Godfrey, Sara, Carolyn Brophy, et. al. Assessing the Ada Design Process and its Implications: a Case Study, GSFC Document SEL-87-004, July 1987.

[Goldberg 83]
Goldberg, Adele and David Robson. Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

[Nelson 86]
Nelson, Robert W. "NASA Ada Experiment -- Attitude Dynamic Simulator," Proceedings of the Washington Ada Symposium, March 1986.

[Parnas 72]
Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, December, 1972.

[Rajlich 85]
Rajlich, Vaclav. "Paradigms for Design and Implementation in Ada," Communications of the ACM, July 1985.

[Seidewitz 86a]
Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," Proceedings of the 1st International Conference on Ada Applications for the Space Station, June 1986.

[Seidewitz 86b]
Seidewitz, Ed and Mike Stark. General Object-Oriented Software Development, GSFC Document SEL-86-002, August 1986.

[Seidewitz 87]
Seidewitz, Ed. "Object-Oriented Programming in Smalltalk and Ada", Proceedings of the Conference on Object-Oriented Programming, Languages, Systems and Applications, October 1987.

[Stark 87]
Stark, Mike and Ed Seidewitz. "Towards a General Object-Oriented Ada Lifecycle," Proc. of the Joint Conference on Ada Technology / Washington Ada Symposium, March 1986.

[Yourdon 78]
Yourdon, Edward and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Yourdon Press, 1978.